Achieving Performance on the Grid

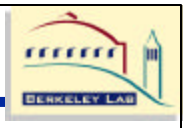
Brian L. Tierney
Dan Gunter

Data Intensive Distributed Computing Group
Lawrence Berkeley National Laboratory

Outline

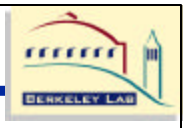
- Intro
  - Why performance analysis is critical for Grid applications
- Monitoring/instrumentation techniques
  - what to monitor
- Analysis Tools
  - NetLogger
- Techniques to optimize application performance in a WAN environment
  - TCP tuning techniques
    - pipechar tool
  - program design considerations
  - async I/O, etc.
- NetLogger Demo

Overview



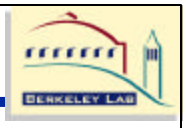
- The Problem
  - When building distributed systems, we often observe unexpectedly low performance
    - the reasons for which are usually not obvious
  - The bottlenecks can be in any of the following components:
    - the applications
    - the operating systems
    - the disks or network adapters on either the sending or receiving host
    - the network switches and routers, and so on
- The Solution:
    - Highly instrumented systems with precision timing information and analysis tools

Bottleneck Analysis



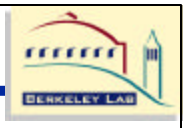
- Distributed system users and developers often assume the problem is network congestion
  - This is often not true
- In our experience tuning distributed applications, performance problems are due to:
  - network problems: 40%
  - host problems: 20%
  - application design problems/bugs: 40%
    - 50% client , 50% server
- Therefore it is equally important to instrument the applications

Monitoring



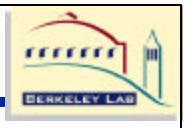
- Monitoring of Applications and Resources is essential in a Grid Environment
- Monitoring Data is needed for:
  - performance analysis
  - performance tuning
  - debugging
- Also for Advanced Grid Services
  - prediction systems (e.g.: NWS)
  - Grid schedulers
  - accounting
  - service verification (e.g.: QoS)

What to Monitor?



- **hosts:** report host stats; e.g.: CPU load, available memory, TCP retransmits
  - could be layered on top of SNMP-based tools, running remotely from the host being monitored
  - could also be used to monitor host configuration information,
    - OS version, software package version, total memory, etc.
- **networks:** perform SNMP queries to a network device
  - e.g.:router or switch.
- **processes**: generate events when there is a change in process status
  - e.g.: process starts, dies normally, dies abnormally
- **storage or I/O:** report storage systems usage of disks or tapes
  - information on block size, access time, seek time, etc.

What to Monitor? (cont.)



- **middleware:** information about middleware services such as directory and authentication servers
- **applications**: sensors can also be embedded inside of applications.
  - generate events if a static threshold is reached (for example, if the number of locks taken exceeds a threshold),
  - upon user connect/disconnect, upon receipt of a UNIX signal, etc.
  - Application sensors can also be used to collect detailed monitoring data about the application to be used for performance analysis.

Network-aware Applications

- To optimally operate over Wide-Area networks, Grid applications must be able to easily adapt to changing network conditions.
  - Must monitor the network (or use a monitoring service)
  - must use that information to:
    - set TCP buffer size
    - set number of parallel streams
    - etc.
  - Much more on this later

The NetLogger Toolkit

NetLogger Toolkit

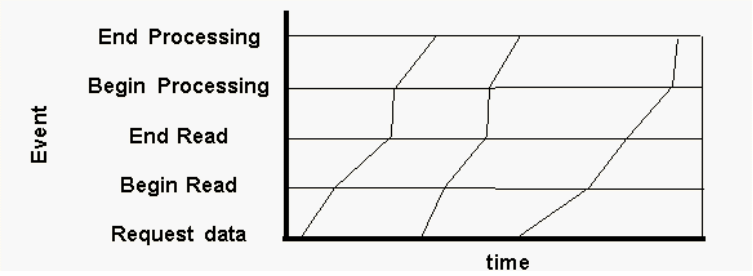
- We have developed the NetLogger Toolkit (short for Networked Application Logger), which includes:
  - tools to make it easy for distributed applications to log interesting events at every critical point
  - tools for host and network monitoring
- The approach is novel in that it combines network, host, and application-level monitoring to provide a complete view of the entire system.
- This has proven invaluable for:
  - isolating and correcting performance bottlenecks
  - debugging distributed applications

NetLogger Components

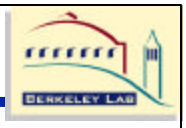
- NetLogger Toolkit contains the following components:
  - NetLogger message format
  - NetLogger client library (C, C++, Java, Perl, Python)
  - NetLogger visualization tools
  - NetLogger host/network monitoring tools
- Source code and binaries are available at:
  - http://www-didc.lbl.gov/NetLogger/
- Additional critical component for distributed applications:
  - NTP (Network Time Protocol) or GPS host clock is required to synchronize the clocks of all systems

Key Concepts

- NetLogger visualization tools are based on time correlated and/or object correlated events.
- NetLogger client libraries include:
  - precision timestamps (default = microsecond)
  - ability for applications to specify an “object ID” for related events, which allows the NetLogger visualization tools to generate an object “lifeline”

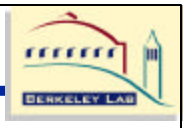


NetLogger Message Format



- We are using the IETF draft standard Universal Logger Message (ULM) format:
  - a list of “field=value” pairs
  - required fields: DATE, HOST, PROG, and LVL
    - —DATE = YYYYMMDDHHSS.SSSSSS
    - —PROG: program name
    - —LVL is the severity level (Emergency, Alert, Error, Usage, etc.)
  - followed by optional user defined fields
  - http://www.ietf.org/internet-drafts/draft-abela-ulm-05.txt
- NetLogger adds this required fields:
  - NL.EVNT, a unique identifier for the event being logged
    - —e.g.: SERVER_IN, VMSTAT_USER_TIME, NETSTAT_RETRANSSEG

NetLogger Message Format

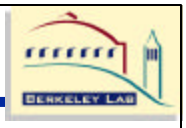


- Sample NetLogger ULM event:

```
DATE=19980430133038.055784 HOST=foo.lbl.gov
  PROG=testprog LVL=Usage NL.EVNT=SEND_DATA
  SEND.SZ=49332
```

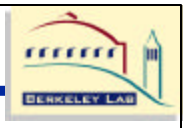
  - This says program named *testprog* on host *foo.lbl.gov* performed event named SEND_DATA, size = 49332 bytes, at the time given
- User-defined data elements (any number) are used to store information about the logged event - for example:
  - NL.EVNT=SEND_DATA SEND.SZ=49332
    - —the number of bytes of data sent
  - NL.EVNT=NETSTAT_RETRANSSEGS NS.RTS=2
    - —the number of TCP retransmits since the previous event

When to use NetLogger



- When you want to:
  - do performance/bottleneck analysis on distributed applications
  - determine which hardware components to upgrade to alleviate bottlenecks
  - do real-time or post-mortem analysis of applications
  - correlate application performance with system information (ie: TCP retransmission's)
- works best with applications where you can follow a specific item (data block, message, object) through the system

When NOT to use NetLogger



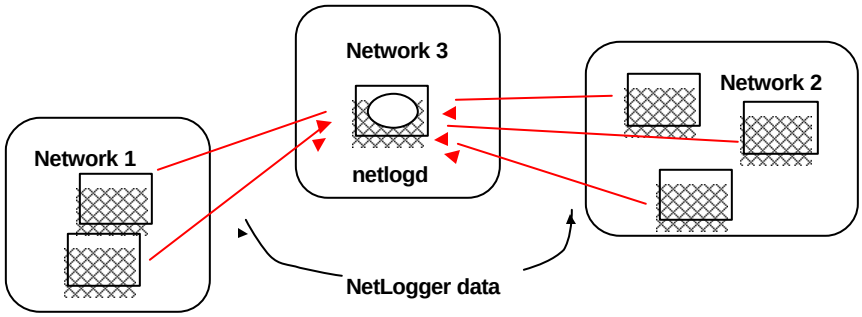
- Analyzing massively parallel programs (e.g.: MPI)
  - Current visualization tools don't scale beyond tracking about 20 types of events at a time
- Analyzing many very short events
  - system will become overwhelmed if too many events
  - we typically use NetLogger to monitor events that take > .5 ms
  - e.g: probably don't want to use to instrument the UNIX kernel

NetLogger API

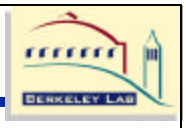
- NetLogger Toolkit includes application libraries for generating NetLogger messages
  - Can send log messages to:
    - file
    - host/port (*netlogd*)
    - syslogd
    - memory, then one of the above
- C, C++, Java, Fortran, Perl, and Python APIs are currently supported

netlogd

- Use *netlogd* to collect NetLogger messages at a central host
  - use to avoid the need to sort/merge several log files from several places

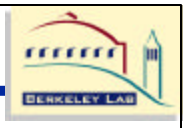


Logging to Memory



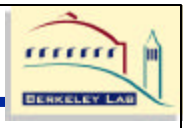
- The NetLogger client library includes an option to buffer log messages in memory:
  - useful if monitoring bursts of events with a duration < 1 ms
- Flushing of events to disk or network will occur:
  - automatically when specified memory block full
  - when calling NetLoggerFlush()
  - when calling NetLoggerClose()
- Size of memory buffer specified by NL_MAX_BUFFER in netlogger.h
  - default = 10,000 messages (typical message size is 128 bytes)

NetLogger API



- Only 6 simple calls:
  - NetLoggerOpen()
    - create NetLogger handle
  - NetLoggerWrite()
    - get timestamp, build NetLogger message, send to destination
  - NetLoggerGTWrite()
    - must pass in results of Unix gettimeofday() call
  - NetLoggerFlush()
    - flush any buffered message to destination
  - NetLoggerSetLevel()
    - set ULM severity level
  - NetLoggerClose()
    - destroy NetLogger handle

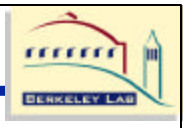
NetLogger Open Call



NLhandle *lp = NULL;
lp = NetLoggerOpen(char *program_name, char *dest_url, int flags);

- program_name: name to be inserted into ULM “program” field
- dest_url: destination of log file; valid URLs formats include:
  - file://path/file
  - x-netlog://host:port
  - x-syslog://localhost
- flags: bitwise “or” of the following:
  - NL_MEM: buffer in memory
  - NL_ENV: destination must be specified by the NL_DEST_ENV environment variable; NetLogger is off if this variable not found
  - NL_APPEND: append to existing log file

NetLoggerOpen() shell environment variables



- Enable/Disable logging:
  `setenv NETLOGGER_ON {true, on, yes, 1}:` do logging
  `setenv NETLOGGER_ON {false, off, no, 0}:` do not do logging
- Log Destination: **`setenv NL_DEST_ENV logging destination`**

Examples:

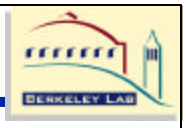
`setenv NL_DEST_ENV file://tmp/netlog.log`
write log messages to file /tmp/netlog.log

`setenv NL_DEST_ENV x-netlog://loghost.lbl.gov`
send log messages to netlogd on host loghost.lbl.gov, default port

`setenv NL_DEST_ENV x-netlog://loghost.lbl.gov:6006`
send log messages to netlogd on host loghost.lbl.gov, port 6006

- NL_DEST_ENV overrides the URL passed in via the NetLoggerOpen() call.

Typical Use



- Using the environment variables, application and middleware developers don't have to worry about command line arguments or middleware APIs to enable/disable logging.
- Example: middleware includes the following call:

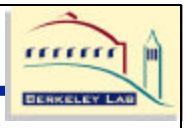
```
NetLoggerOpen("globus", NULL, NL_ENV);
```

  - Default behavior: logging is off
  - If user sets "NL_DEST_ENV" to a valid log destination, then logging will be turned on
- Example: client includes the following call:

```
NetLoggerOpen("my_app", "file://tmp/myapp.log", 0);
```

  - Default behavior: logging is on
  - If user sets: **`NETLOGGER_ON = off`**: Logging is disabled

NetLogger Write Call



- Creates and Writes the log event:

```
NetLoggerWrite(nl, "EVENT_NAME",
  "EVENTID=%d F2=%d F3=%s F4=%.2f", id,
  user_data, user_string, user_float);
```

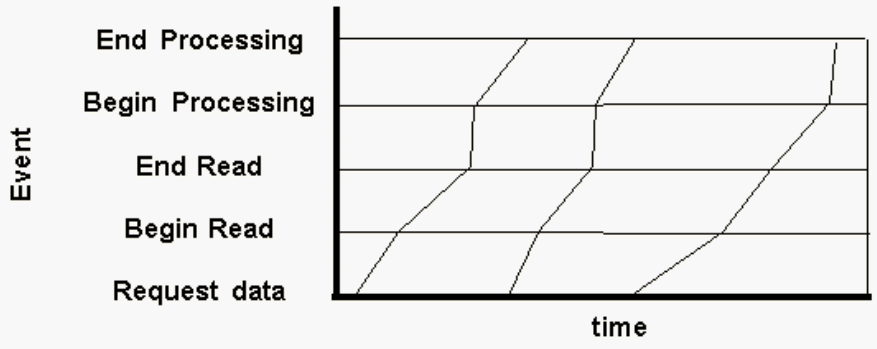
  - timestamps are automatically done by library
  - the "event name" field is required, all other fields are optional
  - this call is thread-safe: automatically does a mutex lock around write call (compile time option)
- Example:

```
NetLoggerWrite(nl, "HTTPD.START_DISK_READ",
  "HTTPD.FNAME=%s HTTPD.HOST=%s", fname,
  hostname);
```

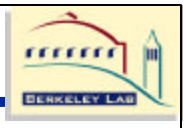
Sample NetLogger Use

```
lp = NetLoggerOpen( progname,
          x-netlog://loghost.lbl.gov, 0);

while (!done)
{
       NetLoggerWrite(lp, "EVENT_START",
                      "TEST.SIZE=%d", size);

       /* perform the task to be monitored */
       done = do_something(data, size);

       NetLoggerWrite(lp, "EVENT_END");
}
NetLoggerClose(lp);
```


NetLogger Event "Life Lines"

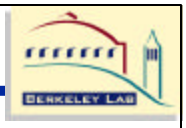


Event ID



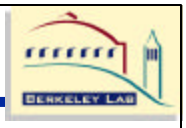
- In order to associate a group of events into a “lifeline”, you must assign an event ID to each NetLogger event
- Sample Event Ids
  - file name
  - block ID
  - frame ID
  - user name
  - host name
  - *combination of the above*
  - etc.

Sample NetLogger Use with Event IDs



```
lp = NetLoggerOpen(progname, NULL, NL_ENV);
for (i=0; i< num_blocks; i++)  {
    NetLoggerWrite(lp, “START_READ”,
      “BLOCK_ID=%d BLOCK_SIZE=%d”, i, size);
    read_block(i);
    NetLoggerWrite(lp, “END_READ”,
      “BLOCK_ID=%d BLOCK_SIZE=%d”, i, size);
    NetLoggerWrite(lp, “START_PROCESS”,
      “BLOCK_ID=%d BLOCK_SIZE=%d”, i, size);
    process_block(i);
    NetLoggerWrite(lp, “END_PROCESS”,
      “BLOCK_ID=%d BLOCK_SIZE=%d”, i, size);
    NetLoggerWrite(lp, “START_SEND”,
      “BLOCK_ID=%d BLOCK_SIZE=%d”, i, size);
    send_block(i);
    NetLoggerWrite(lp, “END_SEND”,
      “BLOCK_ID=%d BLOCK_SIZE=%d”, i, size);
}
NetLoggerClose(lp);
```

NetLogger Host/Network Tools



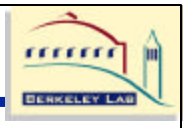
- Wrapped UNIX network and OS monitoring tools to log "interesting" events using the same log format
  - *netstat* (TCP retransmissions, etc.)
  - *vmstat* (system load, available memory, etc.)
  - *iostat* (disk activity)
  - *ping*
- These tools have been wrapped with Perl programs which:
  - parse the output of the system utility
  - build NetLogger messages containing the results

Sample NetLogger System Monitoring Tool

- Example: nl_vmstat -t 60 -d 5000 -m 2 logger.lbl.gov
  - Perl program will exec *vmstat* every 5 seconds for 1 hour, and send the results to *netlogd* on host logger.lbl.gov
  - Generates the following information:
    - CPU usage by User
    - CPU usage by System
- NetLogger Messages:

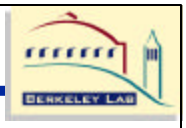
```
DATE=19990706125055.891620 HOST=portnoy.lbl.gov
  PROG=nl_vmstat LVL=Usage NL.EVNT=VMSTAT_USER_TIME
  VMS.VAL=9
DATE=19990706125055. 891112 HOST=portnoy.lbl.gov
  PROG=nl_vmstat LVL=Usage NL.EVNT=VMSTAT_SYS_TIME
  VMS.VAL=5
```

NetLoggerized tcpdump



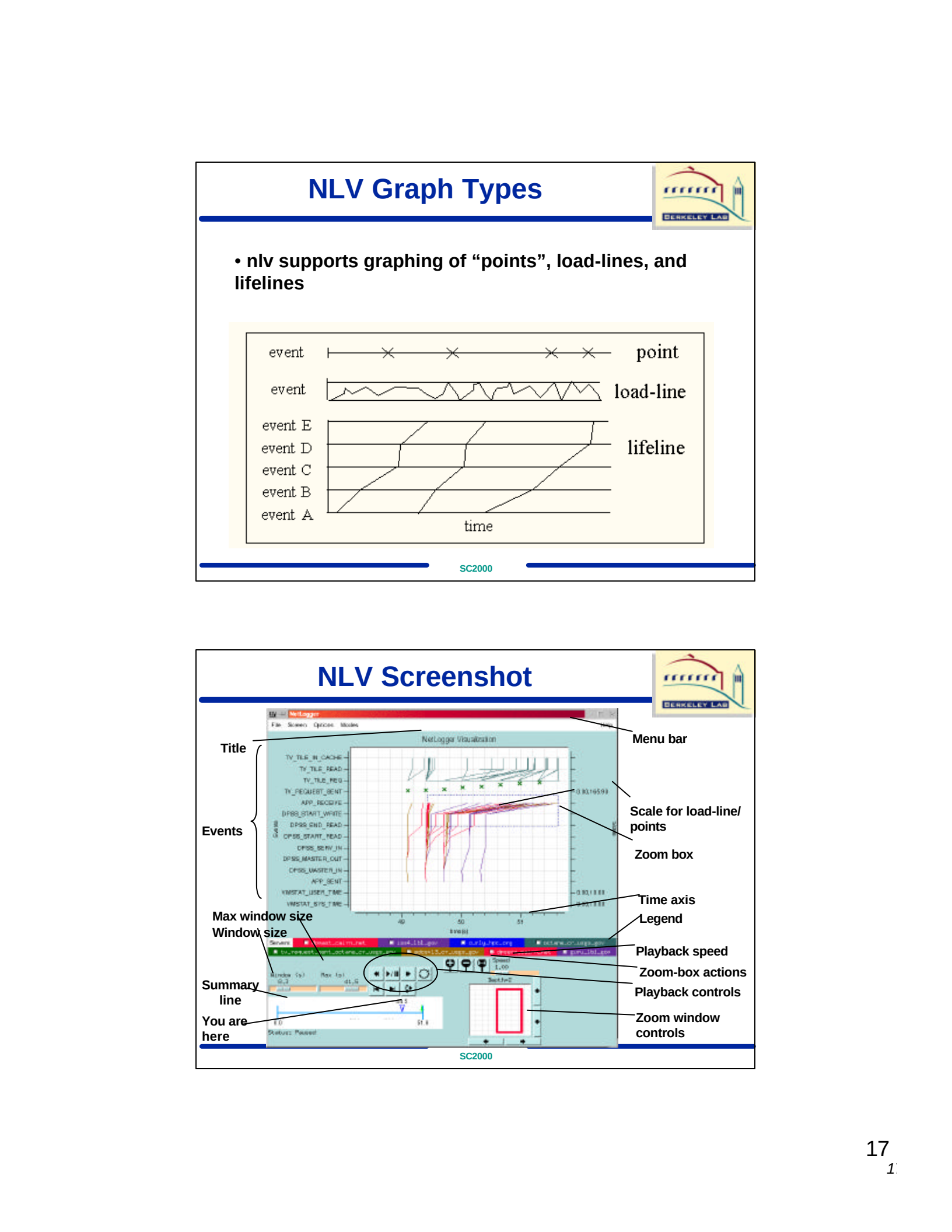
- Precise real-time monitoring of TCP events on a per stream bases
  - TCP retransmits
  - TCP window size
- Example:
  - **tcpdump -A tcp and host piggy.ittc.ukans.edu and port 23**
- Generates the following NetLogger data:
  - **DATE=20000419171039.78654 HOST=piggy.ittc.ukans.edu PROG=tcpdump LVL=ErrorNL.EVNT=TCPD_REXSEG SN=145 SRC_HOST=falcon.cc.ukans.edu SRC_PORT=23 DST_HOST=piggy.ittc.ukans.edu DST_PORT=2800**
- http://www.ittc.ukans.edu/projects/enable/tcpdump

NetLogger Visualization Tools



- Exploratory, interactive analysis of the log data has proven to be the most important means of identifying problems
  - this is provided by *nlv* (NetLogger Visualization)
- *nlv* functionality:
  - can display several types of NetLogger events at once
  - user configurable: which events to plot, and the type of plot to draw (lifeline, load-line, or point)
  - play, pause, rewind, slow motion, zoom in/out, and so on
  - *nlv* can be run post-mortem or in real-time
    - real-time mode done by reading the output of *netlogd* as it is being written

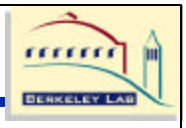
NLV Graph Types



- **nlv supports graphing of "points", load-lines, and lifelines**

NLV Screenshot

NLV Configuration

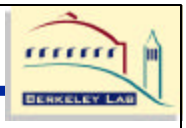


- NLV is very flexible, with many options settable in the configuration file.
- Format:

```
eventset +/-eventset_name {
 { type <line,point,load> }
 { id { list of ULM field names used to determine which
  NetLogger messages get grouped into the same graph
  primitive } }
 { group { list of ULM field names which will be mapped to
  the same color } }
 { val field_name min_val max_val }
 { annotate { list of field names to display in with annotate
  option } }
 { events { list of all event ID's in this lifeline } }
}
```

- Each nlv graph object needs to be defined by an "eventset"
- Events and event-sets both use "+" and "-" to indicate default (i.e. on startup) visibility

Example NLV Configuration



```
# display vmstat info as a "loadline"
eventset +VMSTAT {
{ type load }
# loadline constructed from messages with the same HOST and NL.EVNT
{ id { HOST NL.EVNT } }
# messages with the same HOST get the same color
{ group HOST }
#list of NL.EVNT values in this set_
{ events { +VMSTAT_SYS_TIME +VMSTAT_USER_TIME } }
}

# display netstat TCP retransmits as a "point"
eventset +NETSTAT {
{ type point }
# ignore values outside the range 0 to 999
{ val NS.VAL 0.0 999.0 }
# point constructed from messages from the same HOST and PROG
{ id { HOST PROG } }
# messages with the same HOST get the same color
{ group HOST }
{ events { +NETSTAT_RETRANSSEGS } }
}
```

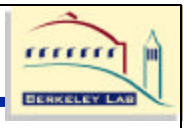
Example NLV Configuration

```
# display server data as a "lifeline"
eventset +SERVER_READ {
{ type line }

# lifeline constructed from messages from the same client
  and server
{ id { CLIENT_HOST DPSS.SERV } }

# messages with the same DPSS.SERV get the same color
{ group DPSS.SERV }

{ events {
  +APP_SENT
  +DPSS_SERV_IN
  +DPSS_START_READ
  +DPSS_END_READ
  +DPSS_START_WRITE
  +APP_RECEIVE }
}
}
```

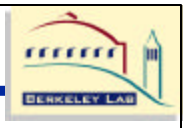

How to Instrument Your Application

- You'll probably want to add a NetLogger event to the following places in your distributed application:
  - before and after all disk I/O
  - before and after all network I/O
  - entering and leaving each distributed component
  - before and after any significant computation
    - e.g.: an FFT operation
  - before and after any significant graphics call
    - e.g.: certain CPU intensive OpenGL calls
- This is usually an iterative process
  - add more NetLogger events as you zero in on the bottleneck

Does NetLogger affect application performance?



- Only if you use it incorrectly, or log too much
- There are several things to be careful of when doing this type of monitoring:
  - If logging to disk, don't log to a nfs mounted disk
    - best to log to /tmp, which may actually be RAM (Solaris)
  - Probably don't want to send log messages to a slow (i.e.: 10BT) or congested network, as you'll just make it worse
    - log to a local file instead

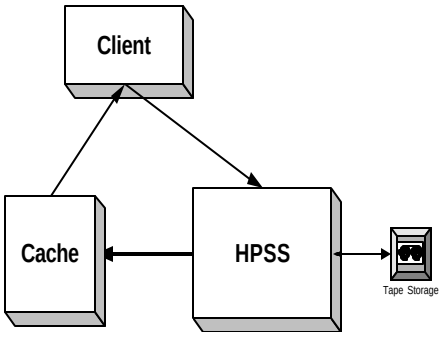


NetLogger Case Studies

Example: HPSS Storage Manager Application

- NetLogger was used to test and verify the results of a Storage Access Coordination System (STACS) by LBNL's Data Management Group
- STACS is designed to optimize the use of a disk cache with an HPSS Mass Storage system, and tries to minimize tape mount requests by clustering related data on the same tape
- NetLogger was used to look at:
  - per-query latencies
  - to show that subsequent fetches of spatially clustered data "hit" in the cache.
- (http://gizmo.lbl.gov/sm/)

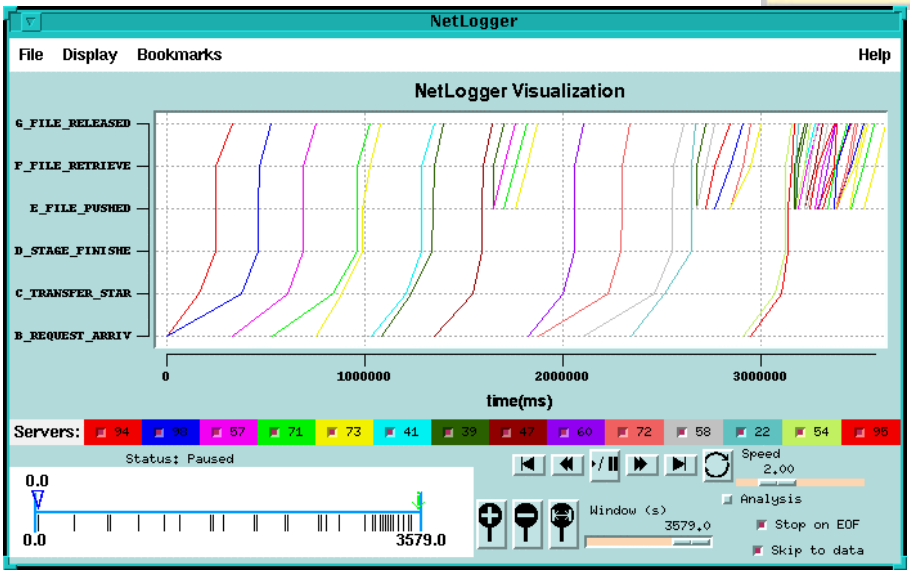
STACS Instrumentation Points



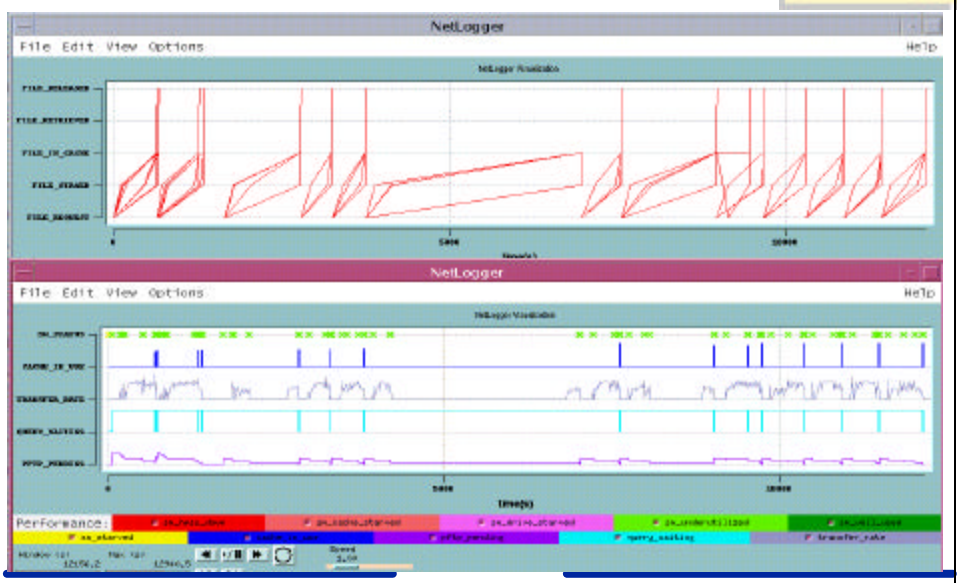
Monitoring Points:

A) request arrives at HPSS
B) start transfer from tape
C) tape transfer finished
D) file available to client
E) file retrieved by client
F) file released by client

NLV for STACS: Tracking File Requests



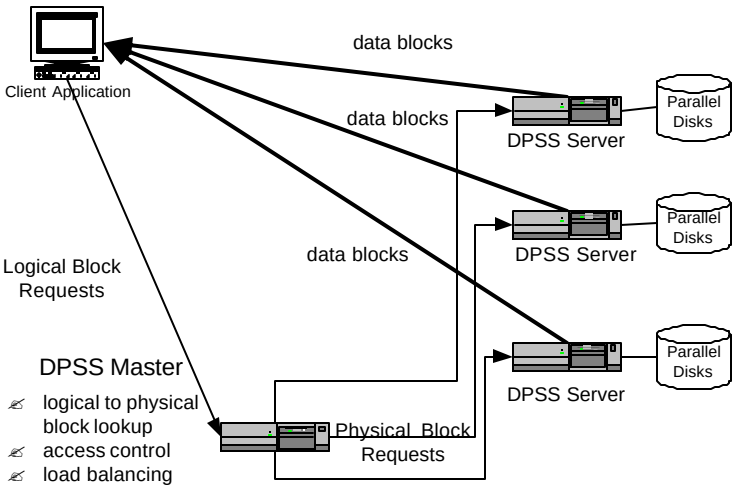
Tracking Files and System Performance



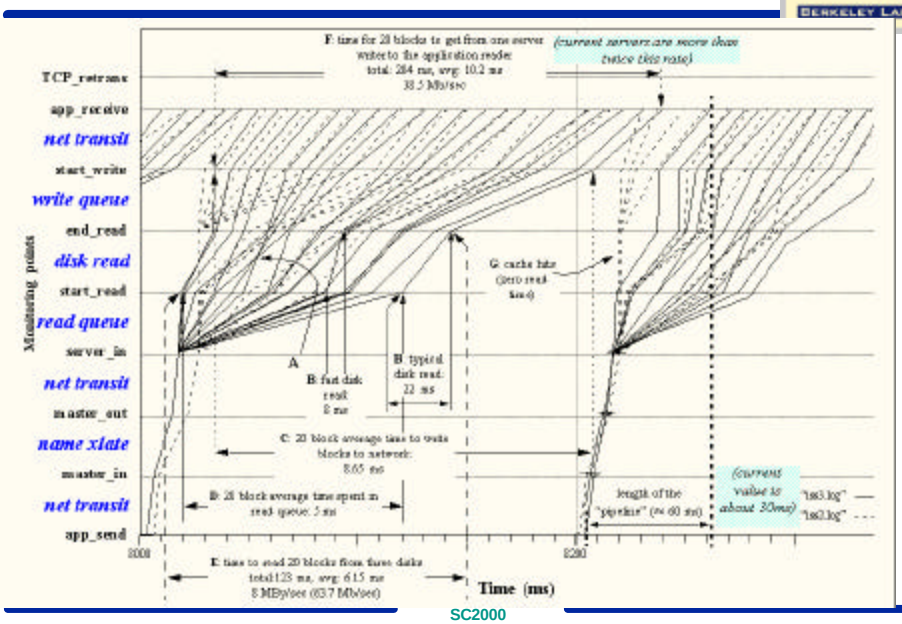
Example: Parallel Data Block Server

- The Distributed Parallel Storage Server (DPSS)
  - provides high-speed parallel access to remote data
  - Unique features of the DPSS:
    - On a high-speed network, can actually access remote data faster that from a local disk
      - —70 MB/sec (DPSS) vs 22 MB/sec (local disk)
    - Only need to send parts of the file currently required over the network
      - —e.g.: client may only need 100 MB from a 2 GB data set
      - —analogous to http model
- NetLogger was used for performance tuning and debugging of the DPSS

DPSS Cache Architecture

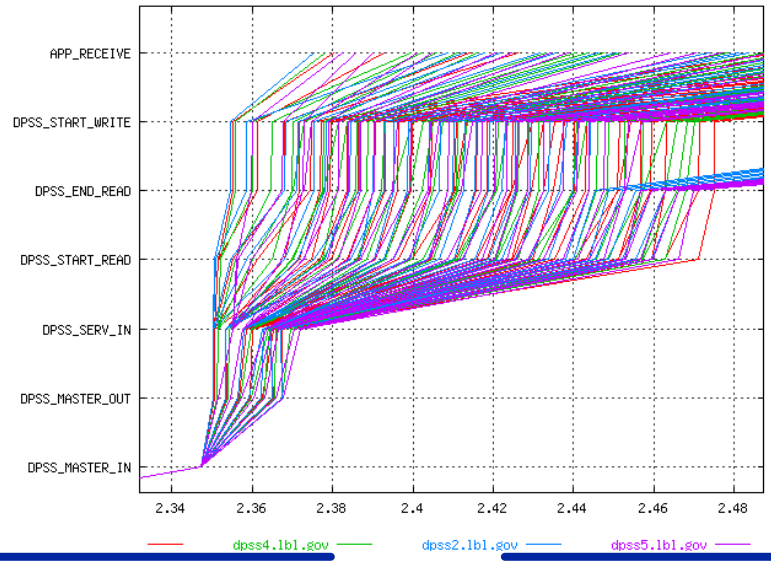


NetLogger Results for the DPSS



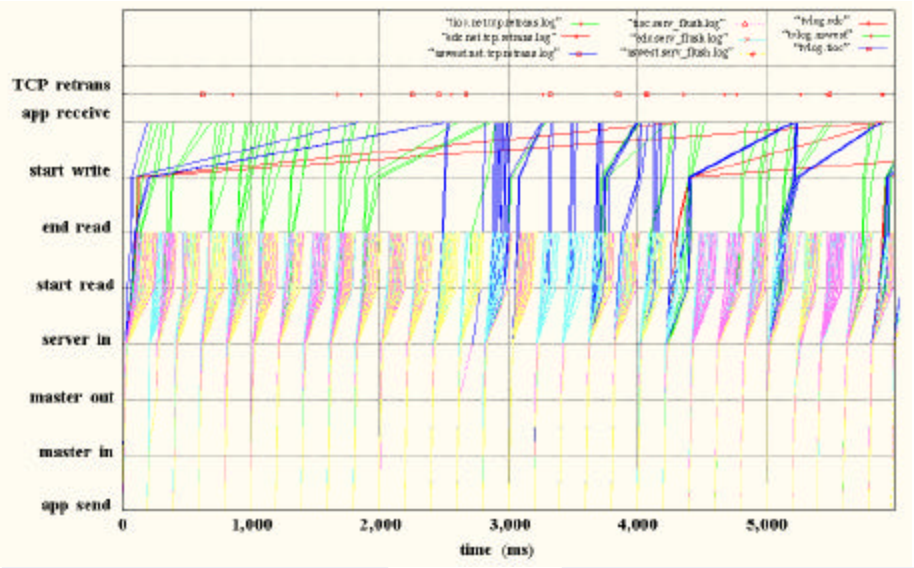
SC2000

NetLogger Results for the DPSS



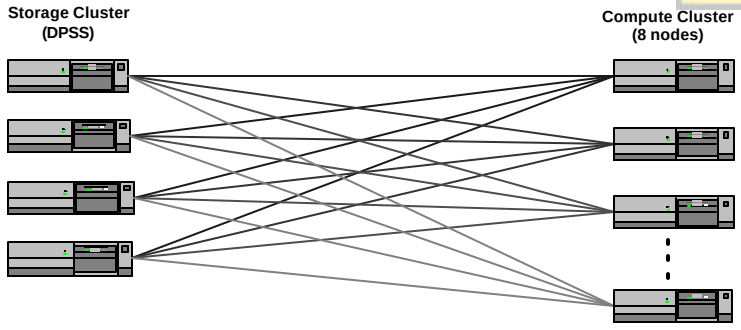
SC2000

NetLogger Results for the DPSS over a WAN

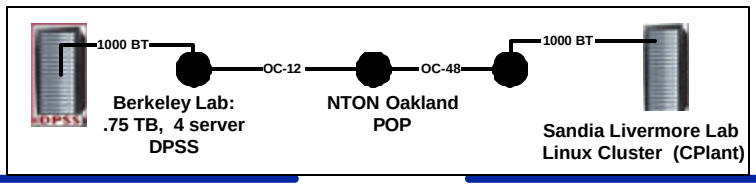


SC2000

DPSS Performance: Used NetLogger for performance tuning

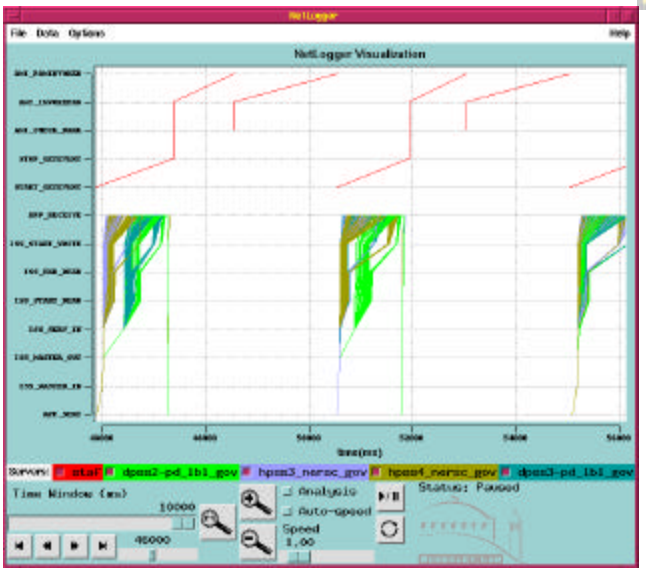


Total Throughput (single dataset to a single cluster application):
570 Mbits/sec (71 MB/sec) on 32 data streams (17 Mbits/sec/stream)

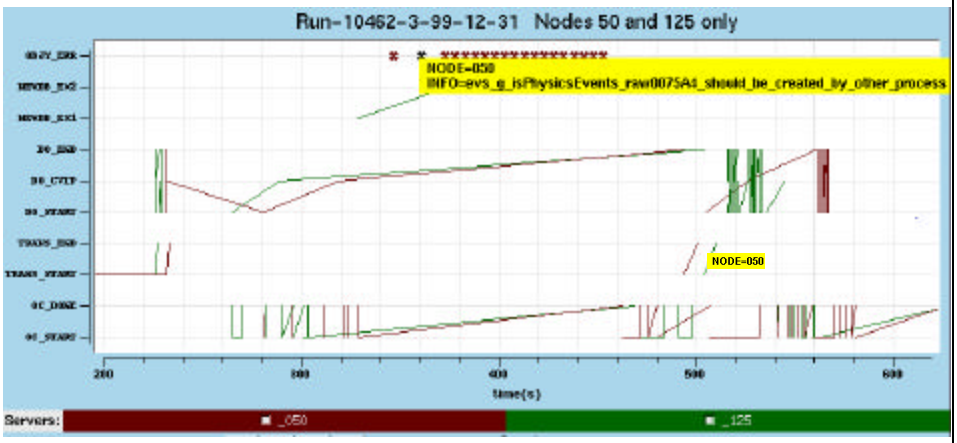


SC2000

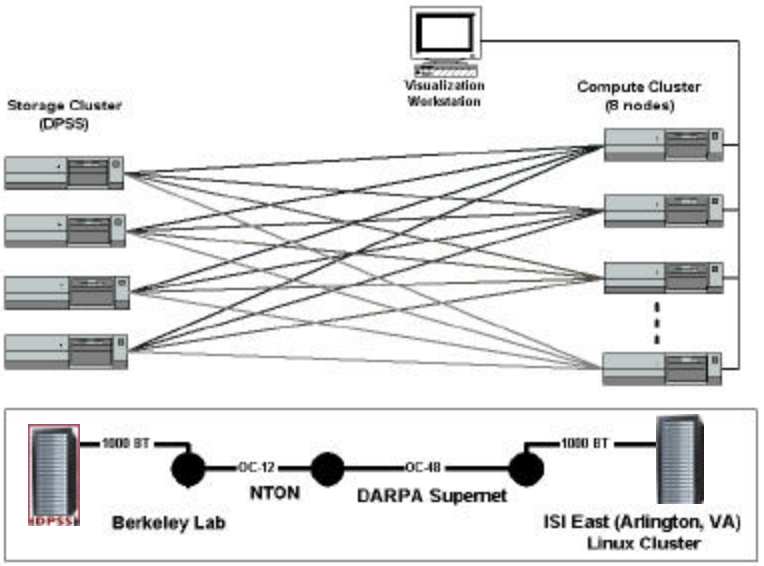
Example: NLV of DPSS with a HENP client



Example: Babar data analysis: 2 nodes with Objectivity Error

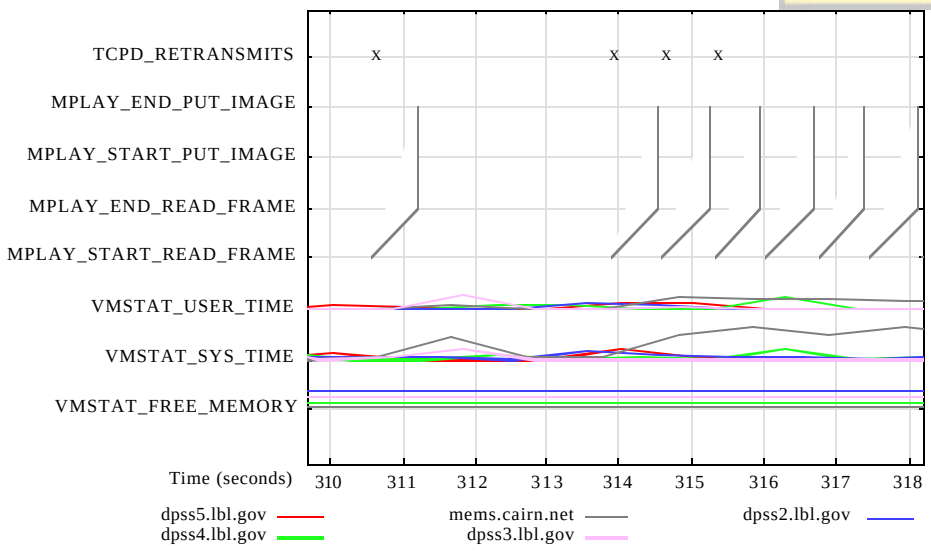


Example: Matisse Project



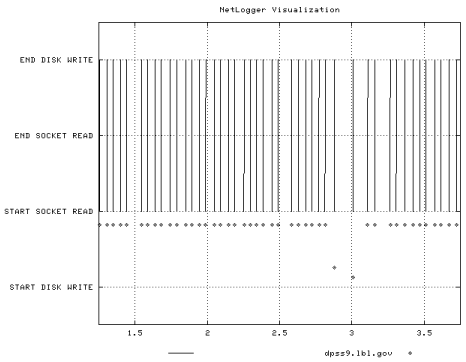
SC2000

Example: Combined Host and Application Monitoring

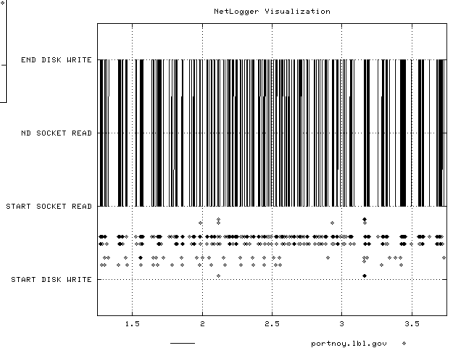


SC2000

Example: NetLogger of ncftp client



- ncftp client on a 10BT ethernet

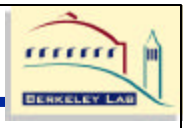


- ncftp client on a 1000BT ethernet

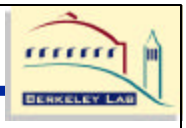
Current/Future NetLogger Work

- Binary format (faster!)
- XML format (slower!!)
- Publish/Subscribe API
  - Producer X
    - NetLoggerPublish(“MONITORING_EVENT_NAME”, ...)
  - Consumer Y
    - NetLoggerSubscribe(X, “MONITORING_EVENT_NAME”, ..)

Getting NetLogger



- Source code and binaries are available at:
  - http://www-didc.lbl.gov/NetLogger
- Client libraries run on all Unix platforms
- Solaris, Linux, and Irix versions of *nlv* are currently supported



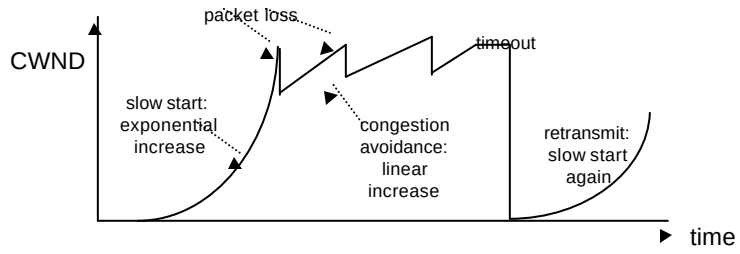
Part 2: Network and TCP Performance Issues

How TCP works: A very short overview

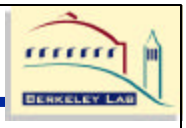
- Congestion window (cwnd)
  - The Larger the window size, higher the throughput
    - Throughput = Window size /Round- trip Time
- Slow start
  - exponentially increase the congestion window size until a packet is lost
    - this gets a rough estimate of the optimal congestion window size
- Congestion avoidance
  - additive increase: starting from the rough estimate, linearly increase the congestion window size to probe for additional available bandwidth
  - multiplicative decrease: cut congestion window size aggressively if a timeout occurs

TCP Overview

- Fast Retransmit: retransmit after 3 duplicated acks (got 3 additional packets without getting the one you are waiting for)
  - this prevents expensive timeouts
  - no need to slow start again
- At steady state, cwnd oscillates around the optimal window size
- With a retransmission timeout, slow start is triggered again

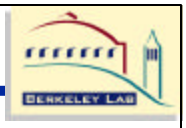


TCP Performance Tuning Issues



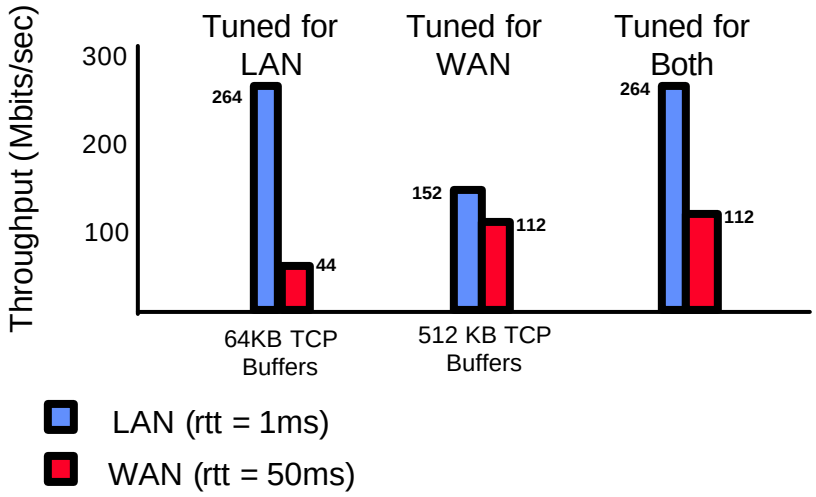
- Getting good TCP performance over high-latency high-bandwidth networks is hard!
- You must keep the pipe full, and the size of the pipe is directly related to the network latency
  - Example: from LBNL to ANL, there is an OC12 network, and the one-way latency is 25ms
    - Bandwidth = 67 MB/sec (OC12 - ATM / IP headers = 539 Mb/s)
  - Need 67 Mbytes * .025 sec = 1.7 MB of data “in flight” to fill the pipe

Setting the TCP buffer sizes



- It is critical to use the optimal TCP send and receive socket buffer sizes for the link you are using.
  - if too small, the TCP window will never fully open up
  - if too large, the sender can overrun the receiver, and the TCP window will shut down
- Default TCP buffer sizes are way too small for this type of network
  - default TCP send/receive buffers are typically 24 or 32 KB
  - with 24 KB buffers, can get **only 2.2%** of the available bandwidth!

Importance of TCP Tuning



TCP Buffer Tuning

- Must adjust buffer size in your applications:

```
int skt, int sndsize;
err = setsockopt(skt, SOL_SOCKET, SO_SNDBUF,
                 (char *)&sndsize,(int)sizeof(sndsize));
```

and/or

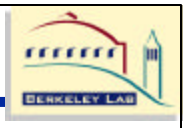
```
err = setsockopt(skt, SOL_SOCKET, SO_RCVBUF,
                 (char *)&sndsize,(int)sizeof(sndsize));
```

- Also need to adjust system max and default buffer
  - Example: in Linux, add to /etc/rc.d/rc.local

```
echo 8388608 > /proc/sys/net/core/wmem_max
echo 8388608 > /proc/sys/net/core/rmem_max
echo 65536 > /proc/sys/net/core/rmem_default
echo 65536 > /proc/sys/net/core/wmem_default
```

- For More Info, see: http://www-didc.lbl.gov/tcp-wan.html

Determining the Buffer Size



- The optimal buffer size is twice the bandwidth*delay product of the link:

  `buffer size = 2 * bandwidth * delay`

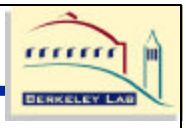
- The ping program can be used to get the delay
  - e.g.: `portnoy.lbl.gov(60)>ping -s lxplus.cern.ch 8192`

```
64 bytes from lxplus012.cern.ch: icmp_seq=0. time=175. ms
64 bytes from lxplus012.cern.ch: icmp_seq=1. time=176. ms
64 bytes from lxplus012.cern.ch: icmp_seq=2. time=175. ms
```

- pipechar or pchar can be used to get the bandwidth of the slowest hop in your path. (see next slides)
- Since ping gives the round trip time (RTT), this formula can be used instead of the previous one:

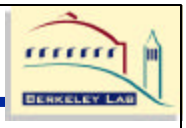
  `buffer size = bandwidth * RTT`

Buffer Size Example



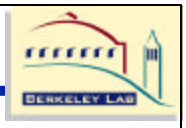
- ping time = 50 ms
- slowest network segment = 10 Mbytes/sec (e.g.: the end-to-end network consists of all 100 BT ethernet and OC3 (155 Mbps)
- TCP buffers should be:
  - .05 sec * 10 = 500 KBytes.
- Remember: default buffer size is usually only 24KB, and default maximum buffer size is only 256KB !

pchar



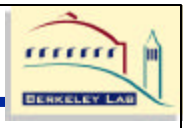
- pchar is a reimplementation of the pathchar utility, written by Van Jacobson.
  - http://www.employees.org/~bmah/Software/pchar/
  - attempts to characterize the bandwidth, latency, and loss of links along an end-to-end path
- How it works:
  - sends UDP packets of varying sizes and analyzes ICMP messages produced by intermediate routers along the path
  - estimate the bandwidth and fixed round-trip delay along the path by measuring the response time for packets of different sizes

pchar details



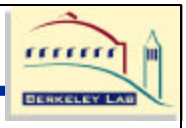
- How it works (cont.)
  - vary the TTL of the outgoing packets to get responses from different intermediate routers.
    - At each hop, pchar sends a number of packets of varying sizes
  - attempt to isolate jitter caused by network queuing:
    - determine the minimum response times for each packet size
    - performs a simple linear regression fit to the minimum response times.
    - This fit yields the partial path bandwidth and round-trip time estimates.
  - To yield per-hop estimates, pchar computes the differences in the linear regression parameter estimates for two adjacent partial-path datasets

Sample pchar output



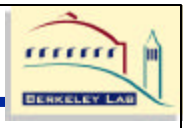

```
pchar to webr.cern.ch (137.138.28.228) using UDP/IPv4
Packet size increments by 32 to 1500
46 test(s) per repetition
32 repetition(s) per hop
 0: 131.243.2.11 (portnoy.lbl.gov)
    Partial loss:      0 / 1472 (0%)
    Partial char:      rtt = 0.390510 ms, (b = 0.000262 ms/B), r2 = 0.992548
                       stddev rtt = 0.002576, stddev b = 0.000003
    Partial queueing:  avg = 0.000497 ms (1895 bytes)
    Hop char:          rtt = 0.390510 ms, bw = 30505.978409 Kbps
    Hop queueing:      avg = 0.000497 ms (1895 bytes)
 1: 131.243.2.1 (ir100gw-r2.lbl.gov)
   Hop char:          rtt = -0.157759 ms, bw = -94125.756786 Kbps
 2: 198.129.224.2 (lbl2-gig-e.es.net)
    Hop char:          rtt = 53.943626 ms, bw = 70646.380067 Kbps
 3: 134.55.24.17 (chicago1-atms.es.net)
    Hop char:          rtt = 1.125858 ms, bw = 27669.357365 Kbps
 4: 206.220.243.32 (206.220.243.32)
    Hop char:          rtt = 109.612913 ms, bw = 35629.715463 Kbps
```


pchar output continued




```
 5: 192.65.184.142 (cernh9-s5-0.cern.ch)
    Hop char:          rtt = 0.633159 ms, bw = 27473.955920 Kbps
 6: 192.65.185.1 (cgate2.cern.ch)
    Hop char:          rtt = 0.273438 ms, bw = -137328.878155 Kbps
 7: 192.65.184.65 (cgate1-dmz.cern.ch)
    Hop char:          rtt = 0.002128 ms, bw = 32741.556372 Kbps
 8: 128.141.211.1 (b513-b-rca86-1-gb0.cern.ch)
    Hop char:          rtt = 0.113194 ms, bw = 79956.853379 Kbps
 9: 194.12.131.6 (b513-c-rca86-1-bb1.cern.ch)
    Hop char:          rtt = 0.004458 ms, bw = 29368.349559 Kbps
10: 137.138.28.228 (webr.cern.ch)
    Path length:       10 hops
    Path char:         rtt = 165.941525 ms, r2 = 0.983821
    Path bottleneck:   27473.955920 Kbps
    Path pipe:         569883 bytes
    Path queueing:     average = 0.002963 ms (55939 bytes)
```

pipechar

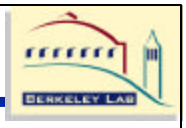


- Problems with pchar:
  - takes a LONG time to run (typically 1 hour for an 8 hop path)
  - often reports inaccurate results on high-speed (e.g.: > OC3) links.
- New tool called *pipechar*
  - http://www-didc.lbl.gov/pipechar/
  - solves the problems with pchar, but only reports the bottleneck link accurately
    - all data beyond the bottleneck hop will not be accurate
  - only takes about 2 minutes to analyze an 8 hop path

pipechar

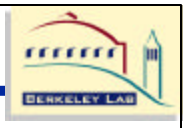
- Like pchar, pipechar uses UDP/ICMP packets of varying sizes and TTL's.
- Differences:
  - uses the jitter (caused by router queuing) measurement to estimate the bandwidth utilization
  - uses a synchronization mechanism to isolate "noise" and eliminate the need to find minimum response times
    - requires fewer tests than pchar/pathchar
  - performs multiple linear regressions on the results

Sample pipechar output



```
>pipechar pdrd10.cern.ch
From localhost:  156.522 Mbps      (157.6028 Mbps)
1: ir100gw-r2.lbl.gov               (131.243.2.1 )
|       157.295 Mbps              <4.9587% BW used>
2: lbl2-gig-e.es.net                (198.129.224.2)
|       159.364 Mbps              <21.5560% BW used>
3: chicago1-atms.es.net             (134.55.24.17)
|       45.715 Mbps               <1.6378% BW used>
4:                                  (206.220.243.32)
|       46.895 Mbps               <1.6378% BW used>
5: cernh9-s5-0.cern.ch              (192.65.184.142)
|       46.330 Mbps               <5.9290% BW used>
6: cgate2.cern.ch                   (192.65.185.1)
|       45.348 Mbps               <10.6760% BW used>
7: cgate1-dmz.cern.ch               (192.65.184.65)
|       46.041 Mbps               <10.1195% BW used>
8: b513-b-rca86-1-gb0.cern.ch       (128.141.211.1)
|       45.411 Mbps    !!!       <23.0134% BW used>
9: b513-c-rca86-1-bb1.cern.ch       (194.12.131.6)
|       46.911 Mbps               <9.3956% BW used>
10: r31-s-rca20-1-gb7.cern.ch        (194.12.129.98)
|       9.954 Mbps *** static bottle-neck 10BT
11: pcrd10.cern.ch                   (137.138.29.237)
```


Other Tools



- iperf : tool for measuring end-to-end TCP/UDP performance
  - http://dast.nlanr.net/Projects/Iperf/
- traceroute: lists all routers from current host to remote host
  - ftp://ftp.ee.lbl.gov/
- tcpdump: dump all TCP header information for a specified source/destination
  - ftp://ftp.ee.lbl.gov/

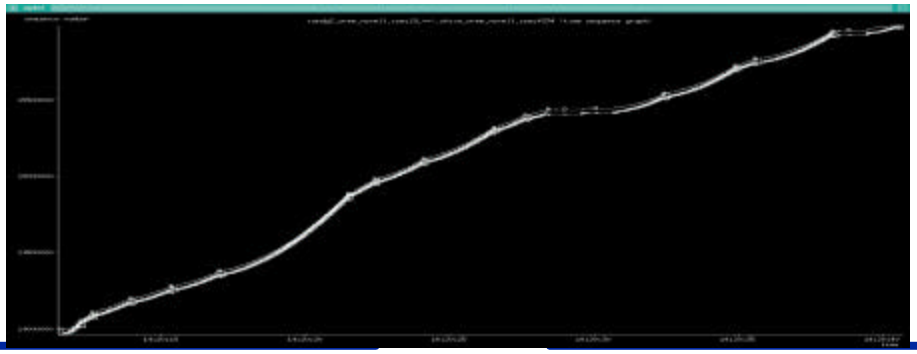
tcptrace

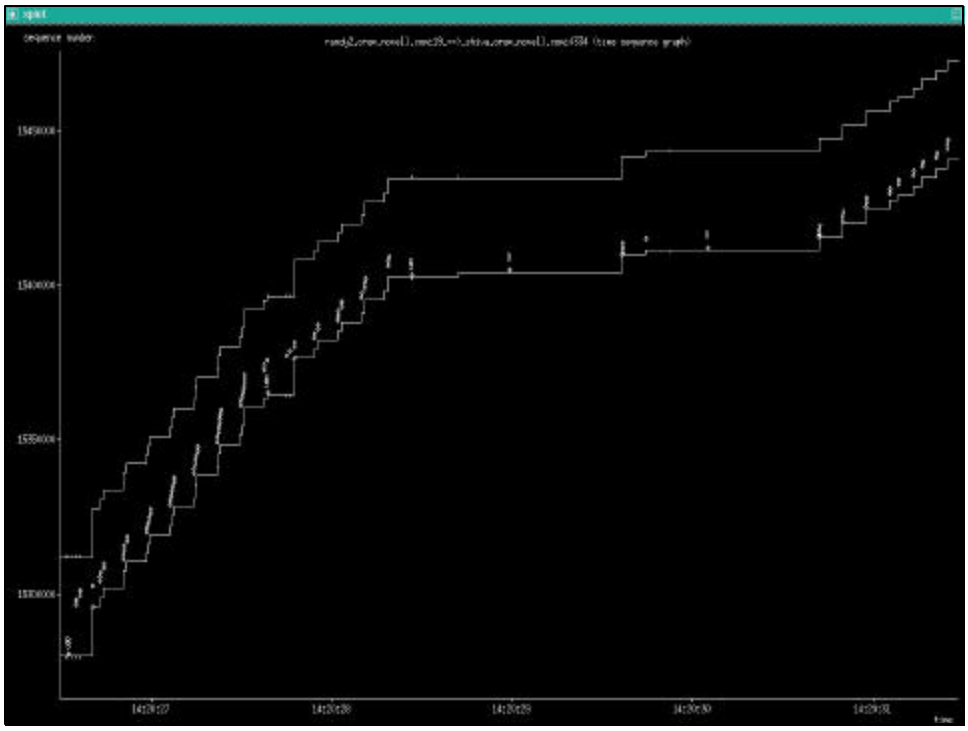
- tcptrace: format tcpdump output for analysis using xplot
  - http://jarok.cs.ohiou.edu/software/tcptrace/
  - NLANR TCP Testrig : Nice wrapper for tcpdump and tcptrace tools
    - http://www.ncne.nlanr.net/TCP/testrig/
- Sample use:

```
tcpdump -s 100 -w /tmp/tcpdump.out host hostname
tcptrace -Sl /tmp/tcpdump.out
xplot /tmp/a2b_tsg.xpl
```


tcptrace and xplot

- X axis is time
- Y axis is sequence number
  - Data packets are indicated with double arrows
  - Window and Acknowledgement numbers as staircases
- Huge range of important scales

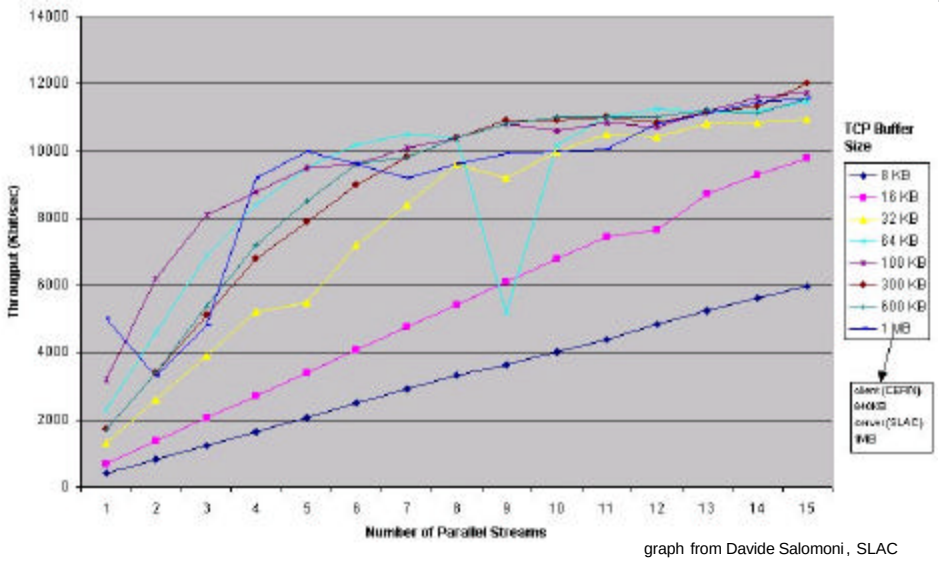




Other Tools

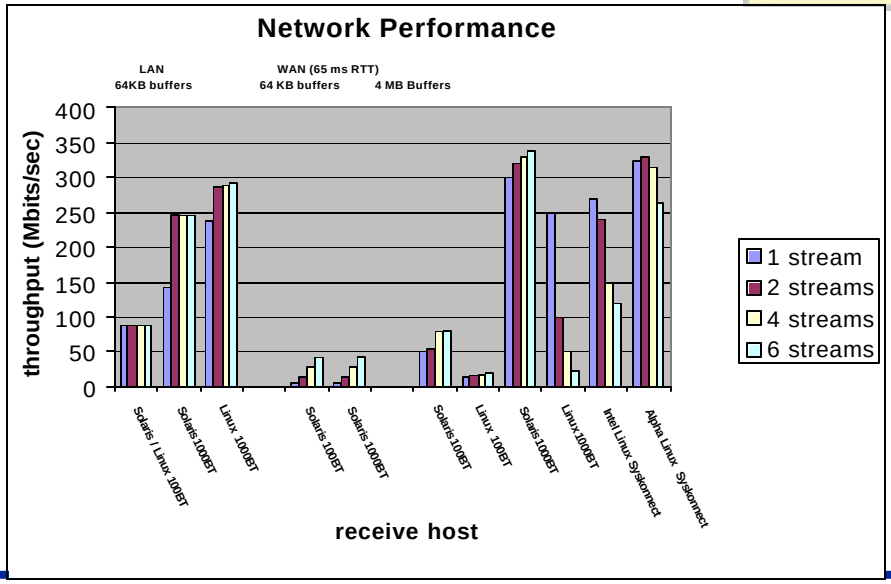
- NLANR Tools Repository:
  - Lots more network analysis tools
  - http://www.ncne.nlanr.net/tools/

Advantage of Parallel Transfers

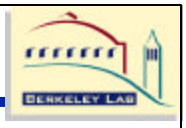


graph from Davide Salomoni, SLAC

TCP WAN Performance: Host Issues

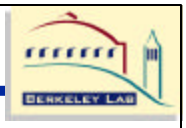


Things to Notice in Previous Slide



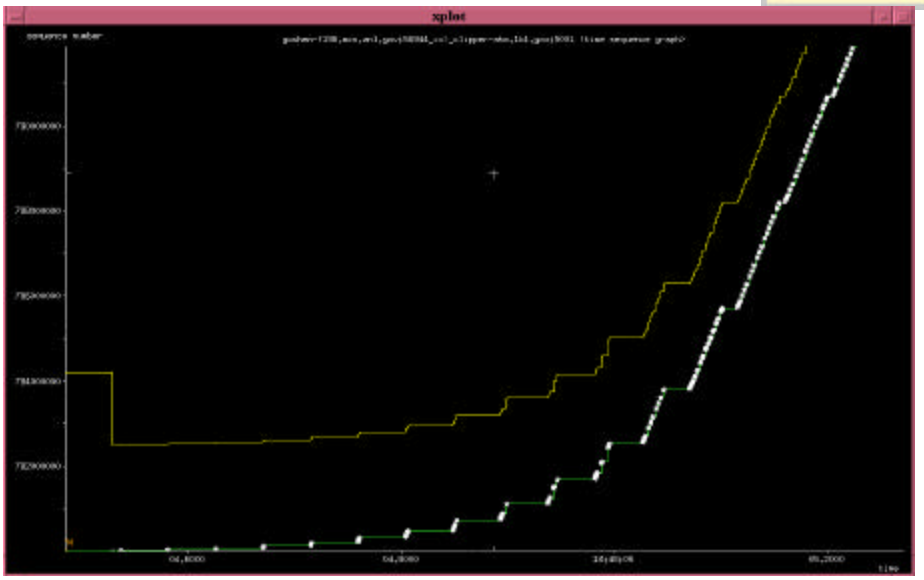
- Parallel Streams help a lot with un-tuned TCP buffers
  - and help a little with large buffers on Solaris
- Problems sending from a 1000BT host to a 100BT Linux host
- Problems sending multiple streams to a 1000BT Linux system, especially with cheap 1000BT hardware

Other TCP Issues

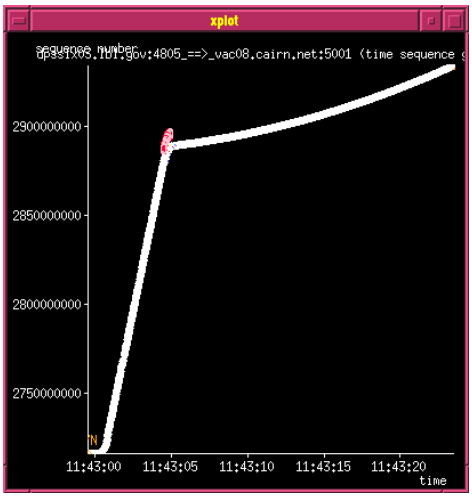


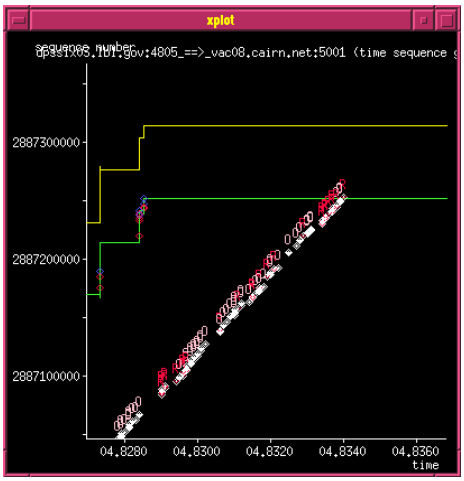
- Things to be aware of:
  - TCP slow-start
    - On the LBL to ANL link, it takes 12 RTT's to ramp up to full window size, so need to send about 10 MB of data before the TCP congestion window will fully open up.
  - router buffer issues
  - host issues

TCP Slow Start

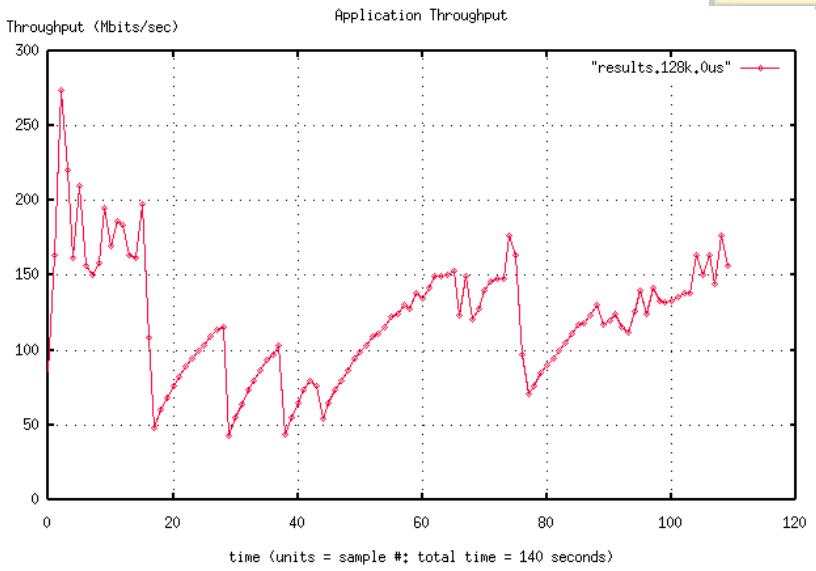


Problems with TCP over NGI-like Networks



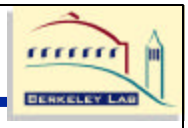


TCP Throughput on DARPA SuperNet



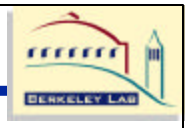
Another Network Performance Issue: Full vs. Half duplex

- A common source of LAN trouble with 100BT networks is that the host is set to full duplex, but the Ethernet switch is set to half-duplex, or visa versa.
- Most newer hardware will auto-negotiate this, but with some older hardware, auto-negotiation sometimes fails
  - result is a working but very slow network (typically only 1-2 Mbps)
  - best for both to be in full duplex if possible, but some older 100BT equipment only supports half-duplex



Application Performance Issues

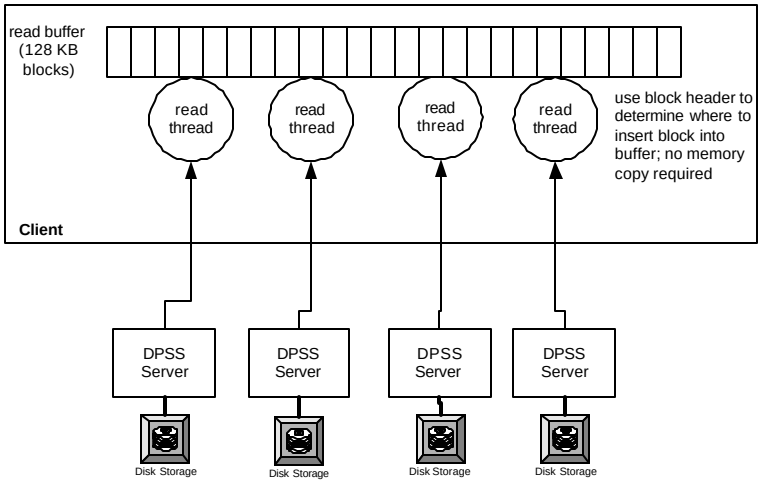
Other Techniques to Achieve High Throughput over a WAN



- Use multiple TCP sockets for the data stream
  - if your receive host is fast enough
- Use a separate thread for each socket
- Keep the data pipeline full
  - use asynchronous I/O
    - overlap I/O and computation
  - read and write large amounts of data (> 1MB) at a time whenever possible
  - pre-fetch data whenever possible
- Avoid unnecessary data copies
  - manipulate pointers to data blocks instead

Threaded Read: dpssRead()

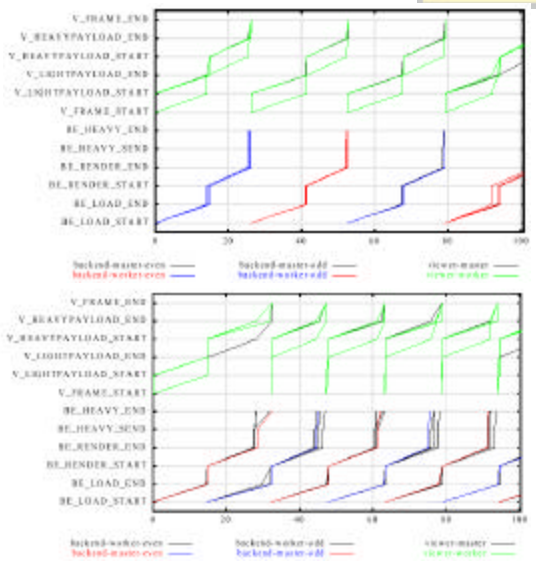
dpssRead(dpss_file_descriptor, char *buffer, int size) ;



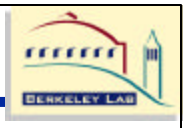
Use Asynchronous I/O

- I/O followed by processing
- overlapped I/O and processing

almost a 2:1 speedup

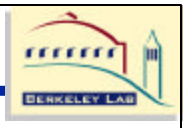


Throughput vs. Latency



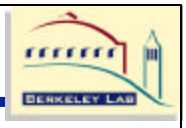
- Most of the techniques we have discussed are designed to improve throughput
- Some of these might even increase latency
  - with large TCP buffers, OS will buffer more data before sending it out.
- Goal of a Grid application programmer
  - hide latency
- However, there are some ways to help latency:
  - use separate control and data sockets
  - use TCP_NODELAY option on control socket
    - But: combine control messages together into 1 larger message whenever possible on TCP_NODELAY sockets

Conclusions



- Tuning Grid Applications is hard!
  - usually not obvious what the bottlenecks are
- Tuning TCP is hard!
  - no single solution fits all situations
    - need to be careful TCP buffer are not too big or too small
    - sometimes parallel streams help throughput, sometimes they hurt

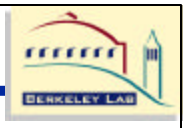
Conclusions



So what to do?

- design your grid application to be as flexible as possible
  - make it easy for clients/users to set the TCP buffer sizes
  - make it possible to turn on/off parallel socket transfers
    - probably off by default
- design your application for the future
  - even if your current WAN connection is only 45 Mbps (or less), some day it will be much higher, and these issues will become even more important

For More Information



Email:bltierney@lbl.gov

http://www-didc.lbl.gov/NetLogger/

- download NetLogger components
- tutorial
- user guide

http://www-didc.lbl.gov/tcp-wan.html

- links to all network tools mentioned here
- sample TCP buffer tuning code, etc.,